

# CSEE4840 Project Design Document

## Battle City

March 18, 2011



### ***Group memebers:***

*Tian Chu (tc2531)*

*Liuxun Zhu (lz2275)*

*Tianchen Li (tl2445)*

*Quan Yuan (qy2129)*

*Yuanzhao Huangfu (yh2453)*

## Introduction:

Our project is to design a video game **Battle City** which was originally developed by Namco in 1985. The player, controlling a tank, must destroy enemy tanks in each level, which enter the playfield from the top of the screen. The enemy tanks attempt to destroy the player's base (represented on the map as a bird, eagle or Phoenix), as well as the human tank itself. A level is completed when the player destroys all 20 enemy Tanks, but the game ends if the player's base is destroyed or the player loses all available lives. The general appearance of the game Battle City looks like this:



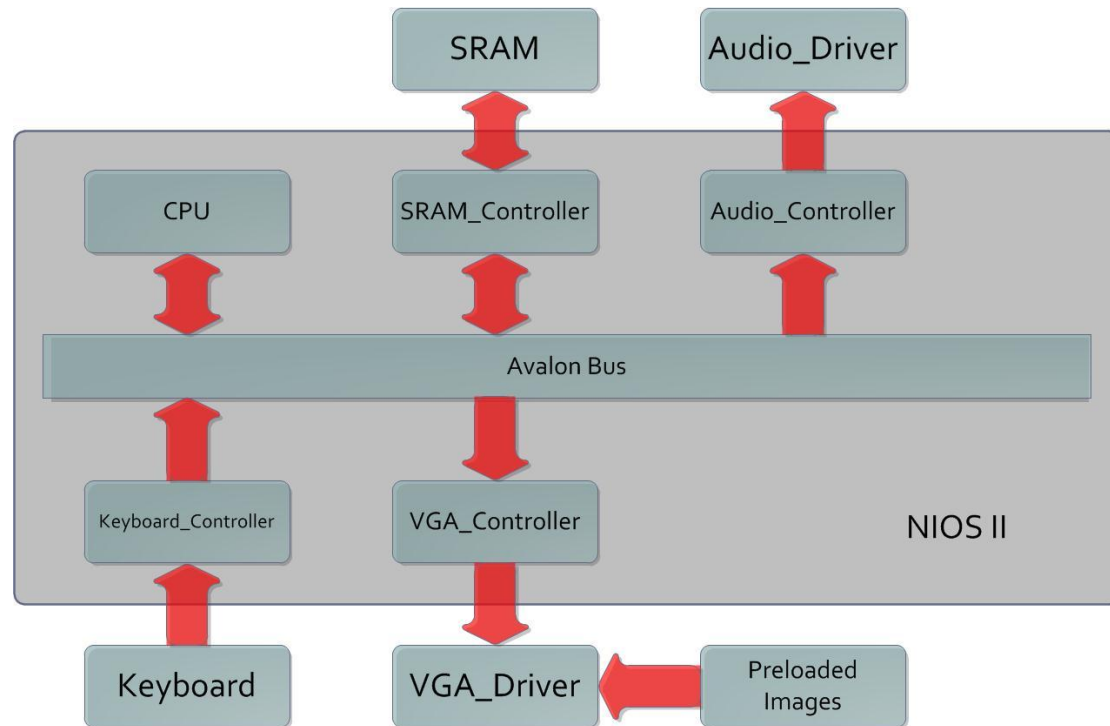
## Challenges:

To implement this project on the DE2 board, we need to integrate the software algorithms with the hardware drivers. The hardware drives the keyboard, VGA monitor and audio decoder. The keyboard receives the user's inputs, like arrow keys to control the moving direction of the tank. The VGA monitor displays the scenario with user's tank, enemies and even the bullet. The audio decoder plays the sound appropriately, like bombing sound when an enemy is destroyed. The real-time video display will be the most challenging part, because the scenario changes all the time and all changes should be synchronized with the software.

The software receives the game player's inputs and translates them into actions of the tank, like moving and fire. At the start of the game, a scenario parser loads the predefined scenario setup and translates it into something that the hardware can understand and displayed properly. During the game playing, the algorithm should control the tanks' movement according to the current scenario setup, and detect the destroying of the enemies. Therefore the multi-tasking may be the most difficult thing the software should handle.

# Architecture:

The following is the basic block diagrams of the whole design architecture, and it shows the connection between the different modules and the interaction between the CPU and hardware drivers.



A brief description of each module is given below:

- **CPU:** loads instructions stored in the SRAM and executes them one by one.
- **SRAM\_Controller:** sends and receives data and instructions between the BUS and SRAM.
- **Audio\_Controller:** receives audio commands from the BUS and translates it to Audio\_Driver.
- **SRAM:** stores data and instructions of the NIOS II needs.
- **Audio\_Driver:** drives the audio decoder with the preloaded sounds.
- **Keyboard\_Controller:** receives keyboard inputs and puts them on the BUS.
- **VGA\_Controller:** receives display data from the BUS and feeds them to the driver.
- **VGA\_Driver:** drives the VGA monitor with the preloaded images.

In the next few sections, we will discuss the keyboard, VGA display, audio play and software algorithm in detail.

## Keyboard:

The keyboard is one of the most important modules in this design, because it provides the only way for the DE2 board to get users' inputs. In this design, we use following keys and their functions are described below:

Keys	Functions
<b>A, D, W, S</b>	Player1's tank moves left, right, up and down respectively
<b>←, ↑, ↓, →</b>	Player2's tank moves left, right, up and down respectively
<b>Space</b>	Player1's tank fires
<b>Enter</b>	Player2's tank fires

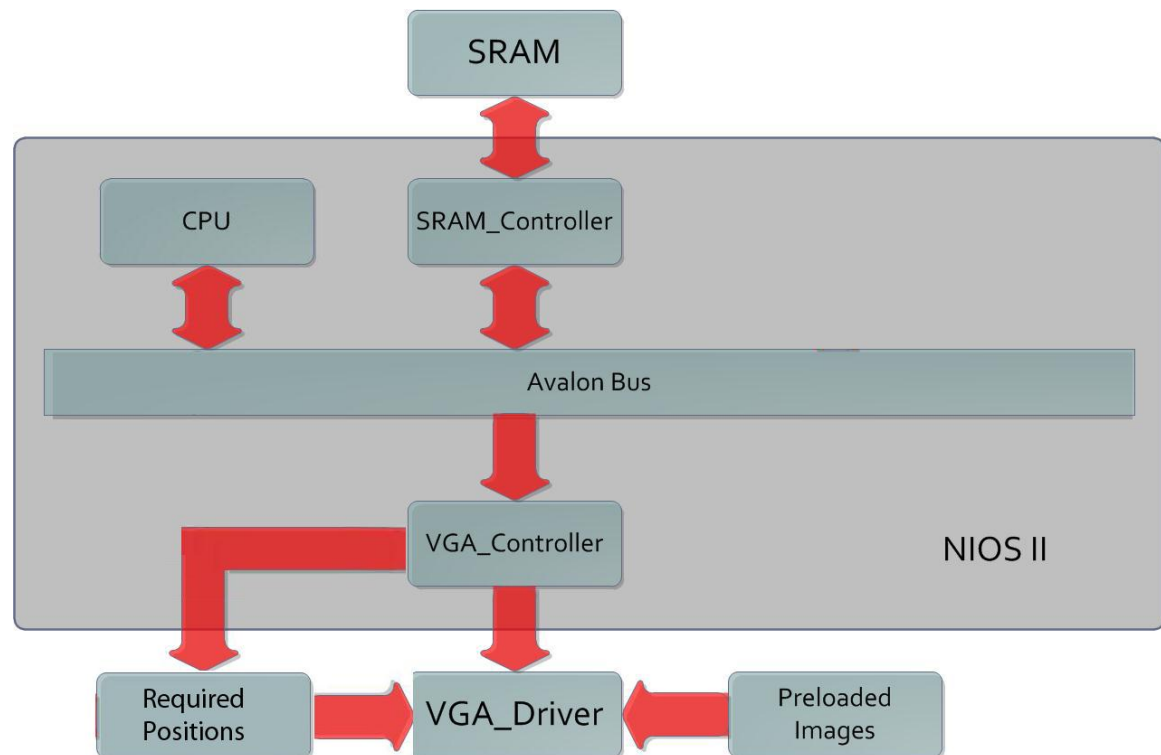
Note: all the functions will be performed repeatedly when the keys are pressed continuously.

## VGA:

The VGA display is the most challenging part of this project, due to the real-time changes of the scenario, like destruction of the walls and hostile tanks. The whole screen is separated into 13X13 squares and each of them is formed by 36X36 pixels. Since all the background items like bricks, river, and concrete are all symmetric in their shapes, we can just store 1/4 of the square's size and repeat them when displaying. The tanks and the bullet are considered as sprites and their images are stored individually in RAMs. In short, the VGA display part of this project should have following functions:

- Loads the scenario setup from the bus sequentially and put the correct images on the screen at proper positions.
- Adjust the tanks and bullets' positions on the screen when positions' update commands received from the bus.
- Handle the overlapping in the game video by using different layers.
- Display animations at proper time and positions when commands are received from the bus, by using image flips.

## VGA Architecture:



VGA display is to draw the game scenario, user interfaces and implement real\_time control. The data and instruction communication is completed via Avlon bus. The VGA display function mainly contains 4 modules: VGA\_Controller, VGA\_Driver, Preloaded Images and Required Positions.

- VGA\_Controller: Receive the CPU instructions into the newly created RAM in FPGA.
- VGA\_Driver: Generate scenario and effects required by CPU, such as move, burst, and disappear.
- Preloaded Images: RAM. Store all the images that may need to display in the game.
- Required Positions: RAM. Store the required position of each image in current scenario.

VGA\_Controller get the CPU instructions of preloaded images and requierd positions then store them into the RAMs. VGA driver reads the memory and implement the real time control to the screen.

## Interfaces:

### PortMaps :

The portmaps are roughly defined below:

- VGA\_Controller(need modification)  
clk : in std\_logic;  
reset\_n : in std\_logic;

```

write      : in  std_logic;
chipselect : in  std_logic;
writedata  : in  unsigned(31 downto 0);
hcenter    : out unsigned(9 downto 0);
vcenter    : out unsigned(9 downto 0)

```

- VGA\_Driver(need modification)

```

reset      : in std_logic;
clk        : in std_logic;           -- Should be 25.125 MHz
TANK_HCENTER : unsigned(9 downto 0); -- Horizontal position (0-800)
TANK_VCENTER : unsigned(9 downto 0); -- Vertical position (0-524)
VGA_CLK,   :                       -- Clock
VGA_HS,    :                       -- H_SYNC
VGA_VS,    :                       -- V_SYNC
VGA_BLANK, :                       -- BLANK
VGA_SYNC : out std_logic;          -- SYNC
VGA_R,     :                       -- Red[9:0]
VGA_G,     :                       -- Green[9:0]
VGA_B : out unsigned(9 downto 0)   -- Blue[9:0]

```

- RAMs

Created automatically by Quartus, during the detail programming progress, the RAMs will be instantiated and utilized by VGA\_Driver.

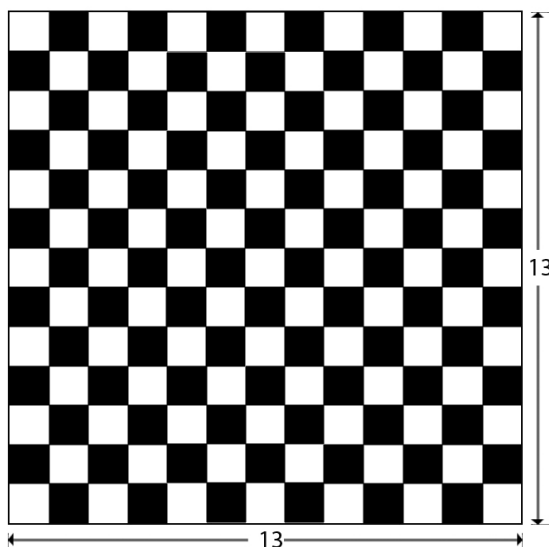
### Instruction Formats

To achieve correct communication, instructions must be aligned between hardware and software. Instruction sets are defined in "data structure". For detail, please refer to Software chapter.

### Image Process:

- User Screen:

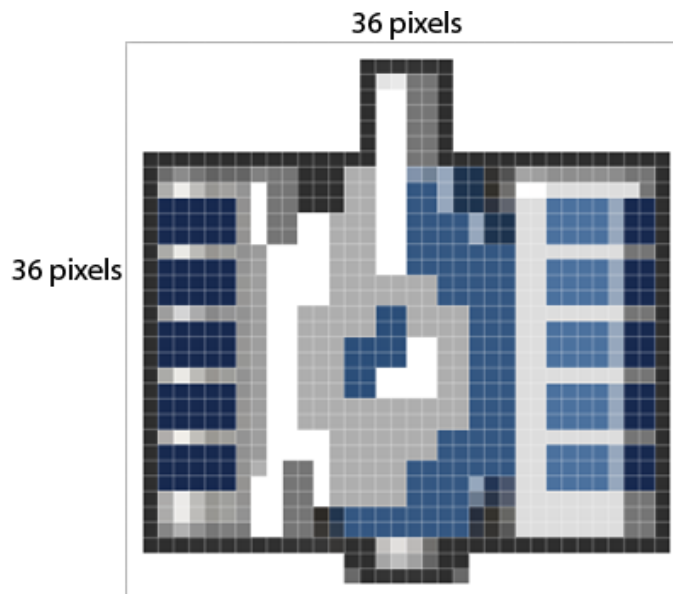
The user screen is divided into 169 squares, each column or row contains 13 squares as indicated below:



The game scenario is constructed by 169 different images that loaded from RAM. VGA\_Controller will determine each image and its position.



The sub-image is formed by 36X36 pixels. The image below is our first tank image.



- Image type:

Two types of images need to be considered for real-time control: static scenario and sprites.

Static Scenario: This type of images are those background images that cannot be moved by player, but may disappear or burst according to events. There are five basic static scenario images:





in the individual RAMs, and being played when the CPU gives the audio controller commands.

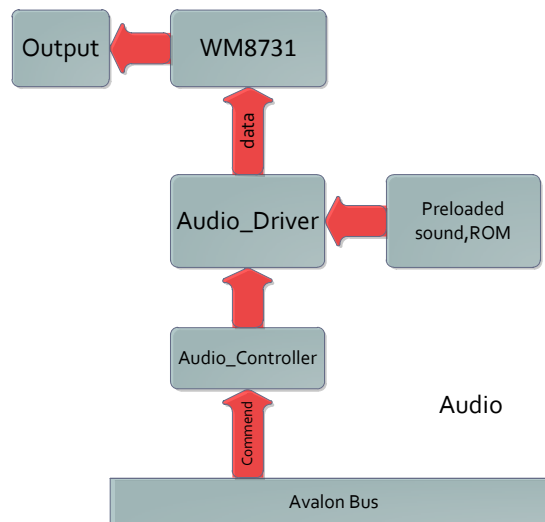
In this game, 8 kinds of sound are used, hence, we plan to use 8-bit control command, whose each bit indicates one kind of sound and '1' means on, '0' means off. Sound and commands are showed in the following table.

Name	Description	Command
<b>Stage Start</b>	welcome music of every stage	10000000
<b>Background</b>	the background sound effect	01000000
<b>Fire</b>	sound of tank firing	00100000
<b>Hit</b>	bullet hits wall or enemy tank	00010000
<b>Movement</b>	sound of tank's moving	00001000
<b>Blast</b>	tank blast or base is destroyed	00000100
<b>get treasure</b>	tank get a treasure	00000010
<b>treasure emerge</b>	treasures appears in the battle field	00000001

We use the following steps to realize the audio function:

- Record 8 kinds of sound from the original game as .wav file.
- Converter wave file to mif file which can be used as a memory initialized file in Quartus.
- Save 8 mif files in rom and the audio driver will read the data from rom when receive commands.
- The audio driver sequential output the data which stored in ROM to DA converter.
- DA converter will output the sound finally.

The block diagram is showed as following:



Some notes:

- Considering the memory space, we may use relatively low sample rate. If the memory space on DE2 board is not enough, we will consider using SDRAM.
- When several sounds are requested to play at same time, the output data will be the sum of the data of each sound effect.

# Software:

Software is definitely the most significant part of this design, since all hardware components are functioning according to the commands received from the software program. To make the game work, the software should have following functions:

- Handles the keyboard input interrupt and translates the make codes and break codes into the corresponding actions of the players' tanks.
- Loads the scenario setup from the SRAM and put them on the bus sequentially for the VGA display module.
- Adjusts and records all tanks' positions and their directions. Due to the existence of the obstacles, some movements should be prevented.
- Adjusts and records all bullets' positions and their directions. When the bullet hits the wall, the background scenario should be updated both in the program and VGA display.
- Generates the enemies at a certain time rate and controls their actions with a random algorithm. The difficulty of the game can be set to different level, and the enemies may be "smarter" in the higher level.

## Data structure:

The whole game includes two kinds of objects—obstacles and sprites. Brick and concrete walls are typical obstacles while the tanks and bullets are sprites. Sprites' positions change with time while the obstacles' don't. To store the obstacles, we use simple integer variables. A 13X13 integer (16-bit) array will be used to store these obstacles, and they will have following format:

X				Y				Blocks				Type			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

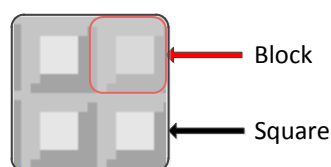
**X:** this four-bit field is used to store the obstacle's horizontal position (0-12 in decimal)

**Y:** this four-bit field is used to store the obstacle's vertical position (0-12 in decimal)

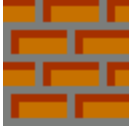

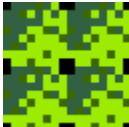
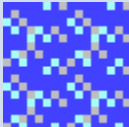

**Blocks:** each obstacle (or square in the screen) is formed by four blocks, and this four-bit field is used to indicate these blocks' existence

**Type:** this four-bit field is used to store the type of the obstacle

This figure illustrates the relationship between an obstacle and its blocks:



The following table illustrates the **Type** value of each kind of obstacles:

	<b>Brick Walls</b>	Type = 0001, Bricks stop tanks and bullets, but they can be slowly chipped away by shooting at them
	<b>Steel Walls</b>	Type = 0010, Steel walls completely stop tanks and bullets. They cannot be destroyed, unless player's tank has collected three stars.
	<b>Trees</b>	Type = 0011, Trees allow tanks and bullets to pass through unchecked. But they partially obscure the view beneath the tree tops.
	<b>Water</b>	Type = 0100, Tanks cannot traverse water, but bullets can safely fly across.
	<b>Ice</b>	Type = 0101, When your tank drives over ice, it will slide a little bit in the direction that it was traveling in when you let go of the button.

Sprites of the game are specified by their positions, types and colors. We will use a 32-bit integer to store a sprite in the Nios program, and it has following format:



<b>X</b>	<b>Y</b>	<b>Types</b>	<b>Color</b>
<b>23-15</b>	<b>14-6</b>	<b>5-2</b>	<b>1-0</b>

**X:** specify the horizontal position of this sprite (0-467 in pixels)

**Y:** specify the vertical position of this sprite (0-467 in pixels)

**Types:** specify the types of the sprites; details are given in the following table

**Color:** specify the color of the sprite, grey = 00, yellow = 01, red = 10 and green = 11

	<b>Player1's tank</b>	<b>Type = 0001</b>
	<b>Player2's tank</b>	<b>Type = 0010</b>

	Basic Tank	Type = 0011
	Fast Tank	Type = 0100
	Power Tank	Type = 0101
	Armor Tank	Type = 0110
	Grenade	Type = 0111
	Helmet	Type = 1000
	Shovel	Type = 1001
	Star	Type = 1010
	Tank	Type = 1011
	Timer	Type = 1100

## Algorithm:

The algorithm of this game is mainly formed by three parts: scenario initialization, bullets trace and keyboard actions. The scenario initialization is executed at the start of the game. The bullets trace is always executed in the main loop, since it has to trace the positions' of bullets and detect the destruction of obstacles, like brick walls, when the bullet hits them. The keyboard action part is executed when a key is pushed, then the player's tank will move in a certain direction for a small step or the player's tank will fire a bullet.

The basic flow-char of the algorithm is plotted in the following figure:

